Interpretação e Compilação de Linguagens de Programação
Sistemas de tipos

26 de Abril de 2013

1 Introdução

A função de interpretação de uma linguagem de programação é normalmente uma função parcial, o que quer dizer que há programas para os quais não é possível computar uma denotação. Neste caso, os programas para os quais a semântica operacional não está definida correspondem a erros de execução. Os erros de execução correspondem normalmente à aplicação de uma qualquer operação primitiva a operandos que não estão no seu domínio. Por exemplo a adição inteira entre um valor inteiro e um fecho (closure), ou a conjunção entre um valor boleano e um valor inteiro, ou a divisão de um número inteiro por zero (0).

As linguagens ditas tipificadas incluem na sua definição um mecanismo de verificação de programas que rejeita programas que potencialmente possam gerar uma classe de erros de execução. Numa linguagem como o Java não é possível executar um programa que por exemplo, invoque um método não existente ou que use um valor inteiro como se fosse um vector ou um qualquer objeto. Não é possível num programa em C♯ ou Pascal, usar um nome que não tenha antes sido declarado e para o qual tenha sido identificado o seu tipo. Estas situações são comuns em linguagens ditas não tipificadas ou tipificadas dinamicamente, como o Ruby, Python, JavaScript entre outras.

O mecanismo de que falamos é um sistema de tipos, um programa que analisa outro programa sem o executar (literalmente) e que avalia a segurança da aplicação das operações que o programa encerra. Tratam-se de ferramentas de análise estática de código, que na realidade são funções de interpretação que trabalham com domínios mais abstratos. Uma característica importante é que o seu processamento termina para todos os programas possíveis dados como entrada.

Se considerarmos o programa seguinte da linguagem CALCState:

```
1 decl
2   a = newvar(0)
```

```
3    b = newvar(2)
4    c = newvar(a > b)
5  in
6    if c then
7      !a := !a + 1;
8       c := 1 < !c
9  end
```

Podemos verificar que há alguns pontos onde o programa pode incorrer num erro de execução. Linha 4, comparação de duas referências, linha 6, condição do if com uma referência, linha 7, afetação de uma não referência. Sendo possível a identificação antecipada destes erros, seria possível produzir o programa sem erros de execução:

```
1  decl
2    a = newvar(0)
3    b = newvar(2)
4    c = newvar(!a > !b)
5  in
6    if !c then
7       a := !a + 1;
8       c := 1 < !a
9  end
```

Definimos então uma função de interpretação de programas abertos da linguagem CALCState para denotações do conjunto *Type*.

$$typecheck : CALCState \times ENV \rightarrow Type$$

Define-se o conjunto *Type* da seguinte maneira (recursiva):

$$Type \triangleq \{IntType, BoolType\} \cup \{RefType(T) \mid T \in Type\} \cup \{None\}$$

Este conjunto contém o elemento *IntType* que representa todos os números inteiros, o elemento *BoolType* que representa todos os valores boleanos, e para cada tipo T, o conjunto de elementos *RefType(T)* que representam todas as referências que contenham valores do tipo T. Assim, a expressão `var(0)` tem a denotação $RefType(IntType)$ (dada pela função typecheck), e a expressão `!x` tem a denotação *IntType*, que corresponde a um valor inteiro durante a execução, num ambiente onde `x` tem a denotação $RefType(IntType)$ e em tempo de execução corresponde a uma variável contendo valores inteiros.

A função de interpretação trabalha num domínio abstrato, de forma sincronizada com o domínio concreto da função de interpretação *eval*. Assim, quando a denotação de uma expressão, dada pela função *typecheck* é *IntType*, a denotação dada pela função *eval* é um número inteiro (do conjunto *Integer*). Uma condição essencial do desenho da semântica operacional e semântica "de tipos" de uma linguagem de programação é esta sincronização. É a condição

```
data Type =
     IntType
  |  BoolType
  |  RefType Type
  deriving (Eq, Show)
```

Figura 1: Tipo de dados

```
typecheck (Num n) env = return IntType

typecheck (Add e e') env =
          do
          t <- typecheck e env
          t' <- typecheck e' env
          case t,t' of
            IntType, IntType -> return IntType
            _,_ -> return None
```

Figura 2: Verificação de tipos

que torna a linguagem (e o seu sistema de tipos) coerente. A função *typecheck* é uma função total, atribuíndo uma denotação a todos os programas da linguagem CALC (ou seja não há erros de execução na computação do tipo).

A denotação *None* corresponde a um programa para o qual não se consegue determinar de forma abstrata o tipo do resultado, ou seja, que durante a sua execução poderá incorrer num erro de execução. Diz-se que a análise estática, feita pelo sistema de tipos é conservativa, o que quer dizer que há programas mal tipificados (com tipo `None`) cuja execução não terminará com um erro.

A definição da função *typecheck* faz-se de maneira composicional, e para algumas construções da linguagem CALC são definidas pela listagem da Figura 2. Note-se que a tipificação da operação `Add` tem como condição os operadores serem de tipo inteiro. Se tal não acontecer, a denotação é `None`.

1.1 Declaração de Identificadores

A semântica de tipos da linguagem com declaração de identificadores é tratada da mesma forma que a semântica operacional. É usado um ambiente que mantém as denotações dos nomes livres da expressão a ser tipificada. Assim, o ambiente utilizado mapeia identificadores em tipos, e a assinatura Haskell da função `typecheck` é:

```
typecheck :: CalcState -> Env Type -> MaybeMessage Type
```

E a semântica para as duas construções `Id` e `decl` são as mostradas na Figura 3.

```
typecheck (Id x) env =
          case find x env of
               Just v -> Result v
               Nothing -> Message ("Identifier "++x++ " not found.")

typecheck (Decl x e e') env =
          do
          t <- typecheck e env
          typecheck e' (assoc env x t)
```

Figura 3: Verificação de tipos com declaração de identificadores.

1.2 Variáveis

A tipificação da extensão desta linguagem para a manipulação de variáveis é mostrada na Figura 4. Nestes casos é visível mais um compromisso que é tomado no nível de interpretação mais abstrato, tornando a análise conservadora. É considerado o invariante da análise que uma variável só pode conter valores de um único tipo.

Note-se que as condições impostas na verificação de tipos são as suficientes para garantir a ausência de erros de execução nestas operações. Nomeadamente que uma afetação só é feita para uma referência, em que o tipo do conteúdo coincide com o tipo previsto para a variável; a desreferenciação também só pode ser feita a uma referência.

2 Expressões de Controlo

Na tipificação das estruturas de controlo (`if`, `while`) (Figura 5) há dois comentários a fazer. O primeiro é que a tipificação de uma expressão condicional (`if`) verifica os dois ramos da expressão (em vez de visitar apenas um como na semântica operacional). Isto deve-se a ser feito sobre numa situação em que não se sabe qual dos ramos vai ser avaliado em tempo de execução. No caso do while, é que a verificação é feita uma só vez, e não é visitado um número indeterminado de vezes como o pode ser na semântica operacional. No caso do if, há uma condição menos óbvia a impôr, que os resultados dos dois ramos sejam do mesmo tipo. Mais uma vez, trata-se de uma aproximação conservadora aos resultados possíveis para a expressão.

3 Tipos funcionais

O tipo dos valores funcionais (closures) descreve quer o tipo do parâmetro que o tipo do resultado. O tipo de uma função funciona como um contrato, se a função for aplicada a um valor do tipo do parâmetro, então o resultado será garantidamente do tipo de resultado atribuído.

```
typecheck (Var e) env =
          do
          t <- typecheck e env
          return (RefType t);

typecheck (Deref e) env =
          do
          t <- typecheck e env
          case t of
             RefType t' -> return t'
             _ -> Message "Dereferencing a non-reference value."

typecheck (Assign e e') env =
          do
          t <- typecheck e env
          t' <- typecheck e' env
          case t of
             RefType t'' | t'' == t' -> return t'
             RefType _ -> Message "Type mismatch on assignment."
             _ -> Message "Assigning a non-reference value."
```

Figura 4: Verificação de tipos com variáveis.

Para determinar o tipo de uma função é necessário tipificar a sua definição. Trata-se da tipificação do corpo da função num ambiente onde o identificador (parâmetro) é associado ao tipo do parâmetro (ver figura 6). A tipificação da aplicação resume-se a garantir que o parâmetro e o argumento coincidem em tipo.

Note-se que o corpo da função é tipificado apenas uma vez, e na sua declaração. Ao contrário da semântica operacional que visita o corpo da função na aplicação. A aplicação é tipificada sem conhecer a implementação concreta da função e apenas o seu tipo.

Note-se que a sintaxe abstrata da definição da função é extendida com uma anotação de tipo. A sintaxe concreta terá que incluir essa anotação, ou seja `fun x : t -> e`.

4 Outros Tópicos

Outros tópicos ainda referidos nas aulas foram a inferência de tipos, com um algoritmo de tipificação associado a um algoritmo de unificação implementado por *union-find*, e ainda a produção de uma AST anotada com os tipos das expressões intermédias de modo a auxiliar no processo de compilação. Os ficheiros que mostram os detalhes podem ser encontrados na página da

```
typecheck (If e e' e'') env =
          do
          t_cond <- typecheck e env
          t_then <- typecheck e' env
          t_else <- typecheck e'' env
          case t_cond of
               BoolType | t_then == t_else -> return t_then
               BoolType -> Message "Type mismatch on conditional branches."
               _ -> Message "Not a Bool value in if condition."

typecheck (While e e') env =
          do
          t_cond <- typecheck e env
          t_body <- typecheck e' env
          case t_cond of
               BoolType -> return BoolType
               _ -> Message "Not a Bool value in while condition."
```

Figura 5: Verificação de tipos com controlo.

cadeira (`infer.ml`, e `CalcStateComp.hs`).

```
typecheck (Fun x t_p e) env =
          do
          t_r <- typecheck e (assoc env x t_p)
          return (FunType t_p t_r)

typecheck (Call e e') env =
          do
          t_f <- typecheck e env
          t_a <- typecheck e' env
          case t_f of
               FunType t_p t_r | t_p == t_a -> return t_r
               _ -> Message "Not a function type value."
```

Figura 6: Verificação de tipos com funções.